Source Code Analysis Tools - Overview

Christoph Michael, Cigital, Inc. [vita[3]]
Steven R. Lavenhar, Cigital, Inc. [vita[4]]

2006-01-27

A security analyzer is an automated tool for helping analysts find security-related problems in software. This article outlines what automated security analyzers can do and provides some criteria for evaluating individual tools.

Introduction

The impetus for security analyzers originally came with the realization that many software vulnerabilities are in reusable library functions, so programs could be scanned to check whether they contain any calls to those functions. This process is more or less equivalent to opening the source code in an editor and searching for the name of vulnerable functions like `strcpy()` and `stat()`.

Modern security analyzers are more sophisticated; they use data- and control-flow analysis to find subtler bugs and to reduce false alarms. They focus on building security in software source code, trying to automate some of the tasks that a human analyst might perform. Unfortunately, these tools are still not capable of replacing a human analyst.

Currently, security analyzers do not unambiguously and flawlessly detect vulnerabilities, and it is therefore erroneous to refer to such a tool as a vulnerability detector. While there are some vulnerabilities that can be detected with high accuracy, others are harder to detect, and, in fact, one can always devise vulnerabilities that are undetectable altogether. Security analyzers are used to make human analysts more efficient; they automate certain mechanical tasks and even certain tasks that are easier for machines than for humans.

However, a security analyzer cannot generate too many false alarms if it is to increase the efficiency of human analysts. Otherwise, too much time is needed for separating the false alarms from the true vulnerabilities. For most security analyzers there is a tradeoff between false alarms (also known as false positives) and missed vulnerabilities (also known as false negatives). Obviously a tool that has fewer false negatives is a good thing because analysts want to catch as many vulnerabilities as possible. On the other hand, false positives make the tool less effective, since much of the analyst's time must be spent weeding them out. It is relatively easy to make a tool more sensitive (decreasing false negatives while increasing false positives) or make it less sensitive (increasing false negatives while decreasing false positives), but most modern security analyzers try to tackle the harder task of decreasing false positives and false negatives at the same time.

While decreasing false positives and false negatives may be the mantra in most of the security scanning industry, there are also some security analyzers that avoid this difficulty by positioning themselves as *detectors of dangerous programming practices*. In other words, they are based on the same philosophy as the classic lint source code checker: it is the developer's job to write code that does not make the security analyzer generate warnings. Like lint, these tools are likely to increase the robustness of the software if they are applied consistently from the start of the development process. On the other hand, applying them to a large, pre-existing codebase is likely to be impractical.

3. daisy:251 (Michael, C. C.)

4. daisy:197 (Lavenhar, Steven)

Finally, security analyzers can be used to generate “badness metrics” [McGraw 04c], giving management and analysts one extra piece of information about the overall quality of the software. Security analyzers are not always perfect in this role either, however. Metrics experts warn against situations where improving the measurements becomes an end in itself, since that practice decouples the metrics from whatever was supposed to be measured in the first place. But the natural tendency is to do just that with the output of a security analyzer: fix the vulnerabilities that the analyzer found. The problem is that a security analyzer can find only some of the security bugs in a piece of software; what percentage it finds is anyone’s guess. Once those bugs are removed, the analyzer will give the software a clean bill of health even if 95% of the original problems are still there. Furthermore, if the security analyzer generates many false alarms, then the work done to address issues it finds may not significantly improve software security.

This point is worth remembering even when a security analyzer is being used to help a human analyst work faster and not as a badness-ometer. No matter how many times an analyzer is run on a given piece of source code, it will always report the same problems. Once those problems are fixed, there is nothing left for the analyzer to say unless it is augmented to provide new types of detection ability (perhaps by the addition of user-specifiable rules). In contrast, a human analyst can find new problems each time he or she examines a piece of source code.

Intended Audience

This document, with its accompanying test programs, is meant for security analysts, and aims to provide an overview of the capabilities of security analyzers. It is also intended to provide a means for evaluating the detection ability of existing tools and their resistance to false alarms. The focus is not on enumerating specific vulnerabilities—that would be impossible as well as potentially misleading—but on categorizing important capabilities of security analyzers and providing the means to evaluate those capabilities.

Scope

This document discusses security analysis tools for software source code. This excludes network-based security analyzers and tools that analyze binary executables, as well as other black box security testing tools. The focus of this document is on analyzers for C/C++ code, though future versions may include analyzers for Java and .NET-based programming languages. We focus on tools that are usable in commercial settings, but the accompanying spreadsheet also lists some academic tools that may or may not be usable in large software development projects. Although it contains criteria for evaluating security analyzers, this document does not include actual evaluations of any tools.

Capabilities of Security Analyzers

This section enumerates some important capabilities that security analyzers have or should have. As far as possible, the section also describes the underlying technologies that provide these capabilities.

Examining Calls to Potentially Insecure Library Functions

The first security analyzers were open-source tools that searched for calls to insecure library functions. Even today this is an important class of vulnerabilities not only because of its prevalence but because of the ease with which hackers themselves can find such flaws.

In some cases it is desirable for a security analyzer to examine the arguments to library functions, since many functions are dangerous only when they are called with certain types of arguments. Some simple tests on function arguments can significantly reduce false alarms. This is especially desirable for certain

vulnerabilities, such as format string vulnerabilities, that are normally avoided by choosing safe arguments rather than by calling a different, non-vulnerable function.

This security-scanning capability can encompass several components:

- **A database of vulnerable library calls** is perhaps the heart of this security scanning technology, but at the same time it is the hardest to evaluate. The vulnerability database must, above all things, be up to date, but an evaluation suite would have to be constantly updated as well to remain relevant.
- **The ability to preprocess source code** is important for C/C++ analyzers, because it lets the analyzer see the same code that will be seen by the compiler. Without this capability there are numerous ways to deceive the analyzer. Many analyzers use heuristics to approximate the functionality of a preprocesser.
- **Lexical analysis** is the process of breaking a program into tokens prior to parsing. Lexical analysis is necessary to reliably distinguish variables from functions and to identify function arguments. These functions can also be performed with heuristics—at the cost of some reliability, however.

Detecting Bounds-Checking Errors and Scalar Type Confusion

A number of vulnerabilities occur in cases where scalar assignments transparently *change* the value being assigned. Examples of this are

- integer overflow: an integer variable overflows and becomes negative
- integer truncation: an integer value is truncated while being cast to a data type with fewer digits
- unsigned underflow: an unsigned integer value underflows and becomes large

When one of these issues results in a vulnerability, it is typically because the affected variable gives the size of a buffer. Typically, such errors can be avoided by placing bounds checks in appropriate places in the code and by type-checking scalars.

To perform robust type checking, an analyzer must be able to parse the code, and the parser must know how to process data types. Typical shortcomings of systems that try to perform type checking with no parser include the inability to distinguish between variables and user-defined types and the inability to determine the types of complex expressions, such as the admittedly obscure C construct

(counter++, y/z),

whose type is the data type of the variable z.

Type checking by itself cannot prevent overflows or underflows. To find potential overflow and underflow problems, an analyzer might keep track of the minimum and maximum values of a variable, or else it might try to ensure that a variable is checked before being used (in a known context) as a buffer length. Detecting potential underflows and overflows is an admittedly challenging problem.[39]

Detecting Type Confusion Among References or Pointers

Type confusion with pointers or references is a common source of bugs and can also result in vulnerabilities unless the type confusion is detected at runtime. C and C++ do not automatically provide such runtime protection.

39. In fact, many interesting static analysis problems are technically impossible due to undecidability, but it is not productive to simply dismiss all attempts to solve such problems. One often finds that a large number of real-world instances of the problem *can* be solved. When we call a static analysis problem “difficult” or “challenging,” we mean that one often seems to encounter challenging instances of that problem in real source code.

In some cases, static type checking can identify reference type confusion. A classic situation that is *not* detected by existing static methods is a cast between incompatible types having a common superclass. If the bad cast is not detected at runtime, it can break the abstraction represented by the data type in question, allowing (for example) methods written for one data type to be applied to a different data type. It is easy to conceive of vulnerabilities that can result from such a situation.

Detecting Memory Allocation Errors

Vulnerabilities involving heap corruption can arise if an attacker is able to overwrite information used to maintain the heap. Usually some of this information is stored together with allocated chunks of memory, e.g., the allocated chunks are stored in a linked list. If an attacker can overwrite one of the links, the operating system can be fooled into writing an arbitrary pointer value to an arbitrary location when it relinks the list after freeing the corrupted chunk. Normally, the heap-maintenance information is not within the range of memory that a program writes to, but a number of circumstances can allow an attacker to corrupt this information. For example:

- a buffer overflow in an allocated chunk of memory
- a double free, where a chunk of memory is freed twice. An attacker might be able to modify the heap-maintenance information if another chunk of memory is allocated between the two frees and if that chunk contains the heap information for the doubly freed chunk.
- a write to freed memory. The affected memory may have been reallocated in such a way that the heap information lies within the range of the freed chunk, allowing attackers to corrupt it.

Memory corruption vulnerabilities can vary from one operating system to the next because the operating systems use different techniques for heap maintenance.

Detecting Vulnerabilities that Involve Sequences of Operations (Control-Flow Analysis)

It is well known that file accesses by a program can create vulnerabilities if done incorrectly. While some vulnerabilities result from the use of inherently insecure functions like stat(), operations also have to be carried out in the right order. For example, a C program that opens a file must first ensure that certain special file descriptors are accounted for and then obtain a file handle to check certain properties of the file before it can access the file contents.

A number of potential vulnerabilities can be introduced when a sequence of operations is carried out incorrectly. For example, the mask governing permissions of newly created files must be set explicitly if a new file may be created, and integer ranges may have to be checked before being used (see Section 2.2) without any modification taking place between the time of check and time of use.

To detect (potential) vulnerabilities associated with incorrectly implemented sequences of operations, security analyzers often look for specific library function calls and print a warning about potential security problems associated with those functions. For example, a call that opens a file might result in a warning about opening files correctly. The problem with this approach is that a warning is triggered regardless of whether the operation in question is being carried out correctly, which causes noise. Most security analyzers support user annotations in the source code that can turn off such warnings. In fact, some analyzers provide an expressive variety of annotations, giving the user some ability to prevent the masking of one security risk by an annotation that was meant for a different risk. Nonetheless, this approach seems slightly unsatisfactory, since the presence of an annotation in the code is not intrinsically connected to the presence or absence of a vulnerability.

Control-flow and data-flow analysis are more robust ways of reducing false alarms. These techniques try to determine whether apparent vulnerabilities can actually be exploited. They also make it possible to

perform entirely new types of analysis.

For example, control-flow analysis can be used to when some potentially dangerous operation must be preceded by precautionary measures, such as closing and reopening standard file descriptors in C before writing to them, or setting default file permissions before creating a new file. Some potential vulnerabilities can also be avoided if the program drops its privileges before carrying out dangerous activities.

Data-Flow Analysis

Security analyzers use data-flow analysis primarily to reduce false positives and false negatives. As a simple (but common) example, many buffer overflows in real code are unexploitable because the attacker cannot control the data that overflows the buffer. Data-flow analysis, in this example, can be helpful in distinguishing exploitable from unexploitable buffer overflows.

The data-flow analysis that seems to be used most often in security-related applications is *taint analysis*. It defines an abstract property of variables called *taint*, which behaves very much like a data type. The most obvious use of taint is to say that a variable is tainted if its value can be influenced by a potential attacker. If a tainted variable is used to compute the value of a second variable, then the second variable also becomes tainted, and so on.

It is also possible to define different types of taint. As a simple example, freeing a pointer could give it a special freed pointer taint, and the security analyzer could detect potential double frees by checking whether a pointer tainted in this way is freed again.

Taint analysis is *static*, similar to static type checking. This tends to make the tainting process overly liberal in the sense that variables may become tainted when they technically should not be.

Pointer-Aliasing Analysis

Pointer aliasing occurs when two pointers point to the same data. The data that would be found by dereferencing one of the pointers can change even though the source code contains no mention of that pointer. This makes static code analysis a greater challenge. *Pointer-aliasing analysis* refers to any static technique that tries to solve this problem by tracking which pointers point to what locations. This analysis can be especially difficult because pointers themselves are just data, and many programming languages allow them to be manipulated in arbitrary ways. For example, imagine a loop that increments the value of a pointer until it points to a space character and then increments the pointer once more (perhaps so it points to the word that comes after the space). Pointer analysis might have to statically answer the question of whether this pointer now points to the tenth character in the string (perhaps another pointer references that location). The problem can be quite difficult if the string is user supplied. Fortunately, some useful pointer-aliasing analysis can still be done without solving difficult or impossible problems.

It is possible to devise vulnerabilities based on pointer aliasing, but the main benefit of pointer-aliasing analysis is that it facilitates data-flow analysis. It is a particularly difficult to solve statically aspect of software analysis, and it is mentioned separately here for that reason.

Customizable Detection Capabilities

Aside from technologies that are meant to reduce false alarms, much of the power of newer security analyzers comes from their ability to support customized detection rules. For example, users of the tool may be able to perform customized data-flow analyses by specifying new types of taint and sources of taint, together with specific rules for how that taint propagates. It may also be possible to specify certain dynamic behaviors that should be checked statically. In short, the available technologies for data-flow

and control-flow analysis allow users of a security analyzer to adapt it for site-specific security policies.

There are also general-purpose static code analysis tools that can be similarly customized, including those that are not explicitly intended as security analyzers but can be used in this capacity by specifying an appropriate set of rules. Because of time and space limitations, this document touches general-purpose tools only in a superficial way. (Furthermore, one could argue that for these tools it is the analyst, not the tool vendor, who provides security analysis capabilities in the first place.)

If customization is planned for a security analyzer, analysts should be aware that the specification and debugging of detection rules can be somewhat time consuming, especially for those who are unfamiliar with the tool. Aside from detection ability (and the ability to support customization in the first place), the *ease* of customization should be given high priority when selecting a security analysis tool.

Overview of the Evaluation Programs

This evaluation suite focuses on analyzers for C/C++ software. It consists of small, simple C/C++ programs, each of which is meant to evaluate some specific aspect of a security analyzer's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms. Below the description of each program is a table that lists the capabilities from Section Capabilities of Security Analyzers[69] that the test program is relevant to.

Programs for Evaluating Detection Ability

- custom_ovf.c: *Buffer overflow using a custom version of the* strcpy() *function*. This buffer overflow is not in the form of a call to a library function. Ability to detect this overflow suggests good data-flow and control-flow analysis capabilities.

Calls to potentially insecure library functions		Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory allocation errors		Pointer aliasing	

- ex_02.c: *Local attacker can cause file-descriptor aliasing.* If this is a setuid program, the attacker can exec() it after closing file descriptor 2. The next time the program opens a file, the file is associated with file descriptor 2, which is stderr. All output directed to stderr will go to the newly opened file. In this example, the attacker creates a symbolic link to the file that is to be overwritten. The name of the link contains the data to be written. When the program detects the symbolic link, it prints an error message and exits (line 32), but the error message, which contains the symbolic link name supplied by the attacker, is written into the targeted file.

  There are several ways to detect this vulnerability, most of which can be characterized as control-flow analysis or data-flow analysis. However, it can also be detected by scanners that vacuously print warning messages for all fprintf statements, which is generally not useful

69. #Capabilities-of-Security-Analyzers

functionality due to false positives. Whether the scanner does this can be determined by running it on ex_02_unex.c

Calls to potentially insecure library functions		Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- ex_03.c: *Race condition while opening a file.* This is a simple race condition, allowing the attacker to change the file named in argv[1] to a symbolic link after it is tested but before the file is opened.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- except.c: *Format-string vulnerability in an error handler.* The catch block in this program contains an exploitable format-string vulnerability. The idea of this test is to see whether the analyzer can track taint through the exception handler. Ideally, the analyzer should report a format string vulnerability on line 32 but not report the unexploitable format string vulnerability in the complementary program unexcept.c below.

  The ability to detect this vulnerability suggests that the analyzer can trace control and data flow through the C++ exception-handling mechanism. However, it can also be detected by printing a warning for all fprintf statements, which is often not useful. Whether the scanner does this can be determined by running it on ex_02_unex.c.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- filedesc.c: *Local attacker can cause file-descriptor aliasing.* If this is a setuid program, the attacker

can exec() it after closing file descriptor 2. The next time the program opens a file, the file is associated with file descriptor 2, which is stderr. All output directed to stderr will go to the newly opened file. In this example, the attacker creates a symbolic link to the file that is to be overwritten. The name of the link contains the data to be written. When the program detects the symbolic link, it prints an error message and exits, but the error message, which contains the symbolic link name supplied by the attacker, is written into the targeted file. This isn't much different from ex_02.c, but the latter program was found on the web claiming to be a secure way of opening files. This program is somewhat simpler and, for some analyzers, might make it easier to tell what the analyzer is printing warnings about.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- macros.c: *A program to test whether a analyzer preprocesses code.*

  A scanner that does not detect the vulnerability, or one that claims a vulnerability is in the second #define rather than in the call to FASTSTRCPY(), probably does not understand C macros.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- overflow.c: *Vulnerability caused by an integer overflow.* In this program, an attacker can supply a large buffer length, which overflows to zero on line 14. Since the subsequent read on line 15 uses the original length value, the read can overflow the buffer.

  Many analyzers will flag the read no matter what, which is useful but doesn't reflect what this program is trying to test. The complementary program notoverflow.c (below) is meant to check whether an analyzer is actually detecting the possible overflow.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among		Data-flow analysis	

pointers			
Memory Allocation Errors		Pointer aliasing	

- signedness.c: *Negative integer turns into large positive string size during cast.* From Secure-Programs-HOWTO/dangers-c.html. In this example, the attacker-controlled number len is read as an integer, and even though there is a test to check whether it's greater than the length of the buffer, a negative value for len will be converted to a large positive value when it gets cast to an unsigned integer in the second call to read.

An analyzer that does not see this vulnerability probably does not understand data types.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- simplefopen.c: *File is opened with no checks at all and can be spoofed by an attacker.* (Often this would be called a race condition as well, but technically it isn't, since the necessary checks are missing entirely.) Early analyzers would be expected to generate warnings on this file because of the fopen(). This test is meant for analyzers that don't warn about anything in ex2_unex.c; it checks whether they just ignore open() calls altogether (ignoring open() isn't what ex2_unex is testing for, needless to say).

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strmacro.c: *This file tries to fool the analyzer by making "strcpy" look like a variable instead of a function.*

  A scanner that fails to find this vulnerability or says the vulnerability is in the #define probably does not preprocess C macros.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors		Control-flow analysis	

and type confusion			
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- strncat_loop2.c: *Buffer overflow caused by a series of strncat()s.*

  This program tests the analyzer's ability to perform data- and control-flow analysis, but an analyzer that vacuously warns of all strncat calls will also create false alarms in strncat.c.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strncat_loop.c: *A series of strncat()s within a loop leads to a buffer overflow*. Technically the buffer in this program has enough room for all the strncat()s, but the programmer forgot to terminate the buffer before the strncat()s begin. Therefore line 7 contains a potential buffer overflow.

  The ability to detect this vulnerability indicates that the analyzer knows something about data on the heap, though the vulnerability could also be detected kludgingly by a rote rule requiring an initialization after a malloc. Once again, the analyzer should not also create false alarms for strncat.c.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strncat_ovf.c: *A strncat() into an unterminated string causes a buffer overflow*. strncpy() doesn't automatically null-terminate the string being copied into. In this example, the attacker supplies an argv[1] of length ten or more. In the subsequent strncat(), data is copied not to buffer[10] as the code suggests but to the first location to the left of buffer[0] that happens to contain a zero byte.

  This program is intended to determine whether an analyzer can keep track of the contents of buffers. As above, the analyzer should not vacuously warn of all strncat calls, as indicated by strncat.c.

Calls to potentially insecure library functions	#	Buffer overflows	#

Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strncat_ovf2.c: *Another strncat into an unterminated buffer*.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strncpy1.c: *Attacker controls argument 2 of strcpy()*.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- truncated.c: *Short buffer allocated because of type mismatch*. This program contains an integer truncation error. Superficially it looks like a safe program even though the buffer length is tainted. It seems as though the buffer is large enough to accommodate whatever data ends up being placed there by the program's read statement. However, the program has a customized malloc() function that takes an int argument, so in reality the malloc on line 3 doesn't always see the same argument as the read on line 18. A value of len larger than 2*MAXINT allows a buffer overflow on line 18.

  This example is somewhat contrived because of the large amount of memory that would have to be allocated for an exploit to succeed. On many architectures, len cannot be greater than 2*MAXINT.

  Checks whether an analyzer understands types.

Calls to potentially insecure library functions	#	Buffer overflows	#
Bounds-checking errors and type confusion	#	Control-flow analysis	

Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- umaskopen.c: *File opened without setting umask*. umask() controls the permissions created by the open call, but the permission mask is passed to the child process in an exec(). If this is a setuid program, the attacker can set a permission mask that makes these files world-writable, but the new file may be a system-critical one. In this program, the programmer uses the umask that existed when the program was exec()ed, but that umask might be controlled by an attacker.

Calls to potentially insecure library functions		Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- umaskopen2.c: *umask() call used incorrectly*. Based on the incorrect statement “umask sets the umask to mask & 0777” in the umask man page. In reality umask sets the mask to 0777 & ~mask, which is also contrary to the convention for chmod that most people are accustomed to. (However, the correct usage is given lower down on the umask man page.) Below, the programmer uses umask() to give the rest of the world full access to the newly created file while denying access to him or herself, which can safely be assumed to be a programming error.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- vptr1.cxx: *Buffer overflow caused by a bad cast*. The principle here is that incorrectly casting a pointer to a C++ object potentially breaks the abstraction represented by that object, since the (non-virtual) methods called on that object are determined at compile time, while the actual type of the object might not be known until runtime. In this example, a seemingly safe strncpy causes a buffer overflow. (In gcc the buffer overflows into the object itself and then onto the stack for this particular program. With some compilers the overflow might modify the object’s virtual table.)

Calls to potentially insecure library functions	#	Buffer overflows	#

Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers	#	Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

Programs for Evaluating Resiliency Against False Alarms

- alias.c: *A test for pointer aliasing analysis*. Since that capability is generally useful only if the analyzer provides some data-flow analysis capabilities, data-flow analysis is needed too. The variable that determines the size of a string copy is untainted, but aliasing analysis is needed to determine this.

Calls to potentially insecure library functions		Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	#

- const_str1.c: *Unexploitable overflow via constant string passed directly to strcpy()*. This program contains a buffer overflow, but the overflowing data isn't controlled by the attacker. Ideally, an analyzer should either not report a buffer overflow associated with this strcpy or at most report a problem with lower severity than a strcpy whose argument is attacker controlled.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- const_str2.c: *Unexploitable overflow by constant string passed indirectly to strcpy()*. This program contains a buffer overflow, but the overflowing data isn't controlled by the attacker. Ideally, an analyzer should either not report a buffer overflow associated with this strcpy() or at most report a problem with lower severity than a strcpy() whose argument is attacker controlled.

  The program is similar to const_str1.c, but it presents a slightly harder problem for the analyzer. In const_str1.c, an analyzer could notice that the argument to strcpy is a constant string by looking for the quote symbol that follows the open parenthesis after the name of the function. In this program,

some sort of data-flow analysis is needed (taint checking should be enough).

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- const_str3.c: *Unexploitable overflow by a constant string stored in a variable*. This is another buffer overflow using a non-user-defined string. Here, the constant string is placed into a variable rather than being passed as a function argument as in const_str2.c. However, taint analysis should still be enough to let the analyzer recognize that the overflowing string is not user controlled.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- ex_02_unex.c: *(Believed) safe file open.* This program ensures that stdin, stdout, and stderr are accounted for and then opens a file, ensuring that access checks are performed on the actual object being opened. The program doesn't set the umask, but that isn't necessary because the umask only affects the permissions of newly created files, and in this program open is called without the O_CREAT flag and therefore will only open a pre-existing file.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- fixedbuff1.c: *Variable-sized buffer that syntactically resembles a fixed-sized buffer*. Many security analyzers generate a warning when they see a fixed-sized buffer. This test program declares a variable-sized buffer based on the length of the string that's going to be copied into it, but it uses a syntax more commonly associated with fixed-sized buffers. It is meant to determine whether an

analyzer detects fixed-sized buffers by looking for square brackets after the variable name or whether it actually parses the declaration.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- fixedbuff2.c: *Variable-sized buffer, syntactically like fixed-sized buffer, whose length is based on a parameter*. This is another variant of a variable-sized buffer being made to syntactically resemble a fixed-sized buffer. It has the added twist that the buffer might be too small if the function useString is called incorrectly, in spite of which there is no buffer overflow here because useString *is* called correctly (and is inaccessible from other source files).

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- fixednamefopen.c: *Believed unexploitable file open, filename with constant path*. This program opens a file with a fixed name in a directory that shouldn't normally be accessible to an attacker. If, for some reason, the attacker has gained write access to /etc, this program could be used to overwrite files in other places, but the vulnerability is less serious than it would be if it opened a file in a directory that's normally writable.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- intarray.c: *Analyzer must resolve typedef to determine the data type of an array*.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation Errors		Pointer aliasing	

- notoverflow.c: *Potential integer overflow is averted by a bounds check*. This program does not contain an integer overflow on line 15 because the length of the variable len is checked. It's meant to complement overflow.c, to check whether buffer overflow warnings for that program are just vacuously triggered by the read() call or if the analyzer is actually spotting the overflow.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- nottruncated.c: *Bounds check averts a potential truncation error*. This program complements truncated.c, which is taken from the Linux secure programming HOWTO. It avoids the integer truncation problem of truncated.c, and it's meant to test whether an analyzer that reports a buffer overflow for truncated.c is doing so vacuously or whether it actually noticed the possible integer truncation.

  In this program, the developer has defined a custom version of the malloc function that takes an int argument and thereby creates the possibility of an integer truncation vulnerability, but bounds checking prevents the malloc from seeing a different length value than the original read.

  This program differs from nottruncated2.c because both mymalloc and read take the original user-controlled size_t len as an argument, but those calls are unreachable for values of len that would cause truncation problems.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- nottruncated2.c: *Almost a truncation error, but not quite.* This program complements truncated.c, which is taken from the Linux secure programming HOWTO. It avoids the integer truncation problem of truncated.c, and it's meant to test whether an analyzer that reports a buffer overflow for truncated.c is doing so vacuously or whether it actually noticed the possible integer truncation.

  In this file, we read a tainted integer and use it to determine the size of a subsequent read of a tainted string. But the buffer receiving the data during the second read is allocated according to user-provided length, and read will only put that many bytes in the buffer, so there should be no overflow.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- signOK.c: *Bounds check prevents a sign error.* This program complements signedness_1.c, where an attacker can create a buffer overflow by specifying a negative number for a buffer length. It checks whether warnings in signedness_1.c are generated vacuously.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion	#	Control-flow analysis	
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strncat.c: *Safe usage of strncpy() and strncat().* This program uses strncpy() and strncat() safely, without introducing a buffer overflow. It is intended to check whether an analyzer warns vacuously about strncpy() and strncat() or actually checks that the buffer sizes are okay and whether the buffer is terminated after the strncpy().

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	
Type confusion among pointers		Data-flow analysis	
Memory Allocation		Pointer aliasing	

Errors			

- strsave.c: *Safe use of strcpy()*. This use of strcpy() ensures that the buffer is large enough to accommodate the string being copied. The data-flow analysis needed to verify this may be too complex to be accomplished with simple taint checking.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- strsave2.c: *Safe use of strcpy(), with the library call enclosed in a wrapper function*. This use of strcpy() ensures that the buffer is large enough to accommodate the string being copied. The data-flow analysis needed to determine whether the strcpy() is safe is somewhat more complex than in strsave.c.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

- unexcept.c: *Format-string vulnerability in an exception handler is unexploitable because of the way that the handler is invoked.* The catch block in this program contains an unexploitable format-string vulnerability. The idea of this test is to see whether the analyzer can track taint through the exception-handling mechanism.

Calls to potentially insecure library functions	#	Buffer overflows	
Bounds-checking errors and type confusion		Control-flow analysis	#
Type confusion among pointers		Data-flow analysis	#
Memory Allocation Errors		Pointer aliasing	

Other Evaluations of Static Security Analyzers

John Wilander and Mariam Kamkar, "A Comparison of Publicly Available Tools for Static Buffer Overflow Prevention[159]," *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, 2002.

Misha Zitser, Securing Software: An Evaluation of Static Source Code Analyzers[160], Master's Thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

M. Zitser, R Lippmann, and T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code," *SIGSOFT Software Engineering Notes 29*, 6 (2004): 97-106.

Michael Zhivich, Tim Leek, and Richard Lippmann, "Dynamic Buffer Overflow Detection[161]," *Workshop on the Evaluation of Software Defect Detection Tools*[162], 2005.

Kendra Kratkiewicz and Richard Lippmann, "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools[163]," *Workshop on the Evaluation of Software Defect Detection Tools*[164], 2005.

Kendra Kratkiewicz, *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*, Master's Thesis, Harvard University, 2005.

The 2005 Workshop on the Evaluation of Software Defect Detection Tools[165] contains a number of papers that may be of interest for evaluating security analyzers, though the workshop itself is broader in scope.

A List of Commercial and Academic Static Security Analyzers

The following list of static security analyzers is incomplete, especially in view of the fact that new tools will undoubtedly appear after the time of this document's writing. However, we have attempted to provide as complete a list as possible of currently available tools. We do not include security analyzers that are unavailable to the general public even though they may be described on the web or in academic papers.

Name	Type	Description
BOON[168]	academic	A model checker that targets buffer-overflow vulnerabilities in C code.
Bugscam[169]	open source	Checks for potentially dangerous function calls in binary

159. http://www.ida.liu.se/%7Ejohwi/research_publications/paper_ndss2003_john_wilander.pdf

160. https://dspace.mit.edu/handle/1721.1/18025

161. http://www.cs.umd.edu/%7Epugh/BugWorkshop05/papers/61-zhivich.pdf

162. http://www.cs.umd.edu/%7Epugh/BugWorkshop05

163. http://www.cs.umd.edu/%7Epugh/BugWorkshop05/papers/62-kratkiewicz.pdf

164. http://www.cs.umd.edu/%7Epugh/BugWorkshop05

165. http://www.cs.umd.edu/%7Epugh/BugWorkshop05

168. http://www.cs.berkeley.edu/%7Edaw/boon/

169. http://sourceforge.net/projects/bugscam

		executable code.
CodeAssure[170]	commercial	General-purpose security scanners for many programming languages.
CodeSonar[171]	commercial	Checks for vulnerabilities and other defects in C and C++.
CodeSpy[172]	open source	Security scanner for Java.
Coverity Prevent[173]	commercial	C/C++ bug checker and security scanner.
Cqual[174]	academic	C Data-flow analyzer using type/taint analysis. Requires some program annotations.
DevPartner SecurityChecker[175]	commercial	Security scanner for C# and Visual Basic
flawfinder[176]	open source	Security scanner for C code.
Fortify Tools[177]	commercial	General-purpose security scanner for C, C++, and Java.
inForce[178]	commercial	Checks for vulnerabilities and other defects in C, C++, and Java.
its4[179]	freeware	Checks for potentially dangerous function calls in C code.
MOPS[180]	academic	Checks for vulnerabilities involving sequences of function calls in C code.
Prexis Engine[181]	commercial	Security scanner for C/C++ and Java/JSP.
Pscan[182]	open source	Checks for potentially dangerous

170. http://www.securesoftware.com/products/

171. http://www.grammatech.com/products/codesonar/overview.html

172. http://www.owasp.org/software/labs/codespy.html

173. http://www.coverity.com/products/prevent.html

174. http://www.cs.umd.edu/%7Ejfoster/cqual/

175. http://www.compuware.com/products/devpartner/securitychecker.htm

176. http://www.dwheeler.com/flawfinder

177. http://www.fortifysoftware.com

178. http://www.klocwork.com/products/inforce.asp

179. http://www.cigital.com/its4/

180. http://www.cs.berkeley.edu/%7Edaw/mops/

181. http://www.ouncelabs.com/prexis_engine.html

		function calls in C code.
RATS[183]	open source	Checks for potentially dangerous function calls in C code.
smatch[184]	open source	C/C++ bug checker and security scanner.
splint[185]	open source	Checks C code for potential vulnerabilities and other dangerous programming practices.

Glossary

access control — Access control ensures that resources are only granted to those users who are entitled to them. [SANS 03]

account harvesting — The process of collecting all the legitimate account names on a system. [SANS 03]

attack — The act of trying to bypass security controls on a system. An attack may be active, resulting in the alteration of data; or passive, resulting in the release of data. Note: The fact that an attack is made does not necessarily mean that it will succeed. The degree of success depends on the vulnerability of the system or activity and the effectiveness of existing countermeasures. [NCSC-TG-004-88]

auditing — The information gathering and analysis of assets to ensure such things as policy compliance and security from vulnerabilities. [SANS 03]

authorization — The approval, permission, or empowerment for someone or something to do something. [SANS 03]

backdoor — A tool installed after a compromise to give an attacker easier access to the compromised system around any security mechanisms that are in place. [SANS 03]

brute force — A cryptanalysis technique or other kind of attack method involving an exhaustive procedure that tries all possibilities, one by one. [SANS 03]

182. http://www.striker.ottawa.on.ca/%7Ealand/pscan/

183. http://www.securesoftware.com/resources/tools.html

184. http://smatch.sourceforge.net/

185. http://splint.org/

buffer overflow	An exploitation technique that alters the flow of an application by overwriting parts of memory. Buffer overflows are a common cause of malfunctioning software. If the data written into a buffer exceeds its size, adjacent memory space will be corrupted and normally produce a fault. An attacker may be able to utilize a buffer overflow situation to alter an application's process flow. Overfilling the buffer and rewriting memory-stack pointers could be used to execute arbitrary operating-system commands. [WebAppSec]
control-flow analysis	Any one of several techniques used to statically trace and characterize the flow of control in software source code.
corruption	A threat action that undesirably alters system operation by adversely modifying system functions or data. [SANS 03]
data-flow analysis	Any one of several techniques used to statically trace and characterize the flow of data in software source code.
defense in depth	The approach of using multiple layers of security to guard against failure of a single security component. [SANS 03]
denial of service	The prevention of authorized access to a system resource or the delaying of system operations and functions. [SANS 03]
Dynamic Link Library (DLL)	A collection of small programs, any of which can be called when needed by a larger program that is running in the computer. The small program that lets the larger program communicate with a specific device such as a printer or scanner is often packaged as a DLL program (usually referred to as a DLL file). [SANS 03]
file descriptor spoofing	An attack where one or more of the three standard C file descriptors, stdin, stdout, or stderr, are closed before executing an application. The next file opened by the application will be assigned one of the standard file descriptors, and output sent to that standard file descriptor will also go to the newly opened file.
format string attack	An exploit technique that alters the flow of an application by using string formatting library features to access other memory space. [WebAppSec]
kernel	The essential center of a computer operating system, the core that provides basic services for all other parts of the operating system. A synonym is nucleus. A kernel can be contrasted with a shell,

	the outermost part of an operating system that interacts with user commands. Kernel and shell are terms used more frequently in UNIX and some other operating systems than in IBM mainframe systems. [SANS 03]
race condition	A race condition exploits the small window of time between a security control being applied and the service being used. [SANS 03]
root	The name of the administrator account in UNIX systems. [SANS 03]
security policy	A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources. [SANS 03]
sensitive information	Sensitive information, as defined by the federal government, is any unclassified information that, if compromised, could adversely affect the national interest or conduct of federal initiatives. [SANS 03]
shell	A UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters. In some systems, the shell is called a command interpreter. A shell usually implies an interface with a command syntax (think of the DOS operating system and its “C:>” prompts and user commands such as “dir” and “edit”). [SANS 03]
stack mashing	Stack mashing is the technique of using a buffer overflow to trick a computer into executing arbitrary code. [SANS 03]
symbolic links	Special files that point at another file. [SANS 03]
tamper	To deliberately alter a system’s logic, data, or control information to cause the system to perform unauthorized functions or services. [SANS 03]
vulnerability	A flaw or weakness in a system’s design, implementation, or operation and management that could be exploited to violate the system’s security policy. [SANS 03]

References for the Glossary

[NCSC-TG-004-88]	http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-004
[SANS 03]	The SANS Institute. *SANS Glossary of Terms Used in Security and Intrusion Detection*.

http://www.sans.org/resources/glossary.php (2003).

[WebAppSec] http://www.weppsec.org/projects/glossary

General References

[Aleph 96] Aleph One. "Smashing the Stack for Fun and Profit[190]." *Phrack Magazine 7*, 49 (1996): File 14 of 16.

[Anderson 96] Anderson, Robert H. & Hearn, Anthony C. *An Exploration of Cyberspace Security R&D Investment Strategies for DARPA: The Day After... in Cyberspace II*. Rand Corporation. MR-797-DARPA, 1996.

[Anderson 01] Anderson, Ross. *Security Engineering*. New York, NY: John Wiley & Sons, 2001.

[AUSCERT 96] AusCERT. *A Lab Engineer's Check List for Writing Secure Unix Code*. Australian Computer Emergency Response Team, 1996.

[Bellovin 94] Bellovin, Steven M. *Shifting the Odds--Writing (More) Secure Software*. Murray Hill, NJ: AT&T Research, 1994.

[Boehm 81] Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Boehm 87] Boehm, Barry W. "Improving Software Productivity." Computer 20, 9 (September 1987): 43-57.

[Boehm 88a] Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *Computer 21*, 5 (May 1988): 61-72.

[Boehm 88b] Boehm, Barry W. & Papaccio, Philip N. "Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering *14*, 10 (October 1988): 1462-1477.

[Bishop 02] Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley Professional, 2002.

[CERT 96] CERT/CC. *CERT Survivability Project Report*. CERT Coordination Center, 1996.

[Chess 04] Chess, Brian & McGraw, Gary. "Static Analysis for Security." *IEEE Security and Privacy 2*, 6 (December 2004): 76-79.

190. http://www.phrack.org/phrack/49/P49-14

[Clements 02] Clements, Paul; Bachmann, Felix; Bass, Len; Garlan, David; Ivers, James; Little, Reed; Nord, Robert; & Stafford, Judith. *Documenting Software Architectures: Views and Beyond*. Boston, MA: Addison-Wesley, 2002.

[Cowan 98] Cowan, Crispin; Beattie, Steve; Finnin Day, Ryab; Pu, Calton; Wagle, Perry; & Walthinsen, Erik. "Protecting Systems from Stack Smashing Attacks with StackGuard," 119-129. *Proceedings of the 1998 Usenix Security Conference*, 1998.

[Cowan 99] Cowan, Crispin; Wagle, Perry; Pu, Calton; Beattie, Steve; & Walpole, Jonathan. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." *Proceedings of DARPA Information Survivability Conference and Expo (DISCEX)*, 1999.

[Demarco 03] Demarco, Tom & Lister, Timothy. *Waltzing With Bears: Managing Risk on Software Projects*. New York, NY: Dorset House Publishing Company, 2003.

[Du 98] Du, Wenliang. "Categorization of Software Errors That Led to Security Breaches[191]." *Proceedings of the 21st National Information Systems Security Conference*. Crystal City, Virginia, Oct. 6-9, 1998.

[Fenton 96] Fenton, Noramn E.. & Pfleeger, Shari Lawrence. *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. New York, NY: International Thomson Computer Press, 1996.

[Garfinkel 03] Garfinkel, Simson; Spafford, Gene; & Schwartz, Alan. *Practical Unix & Internet Security*, 3rd ed. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

[Gilb 98] Gilb, Tom. *Principles of Software Engineering*. Workingham, England: Addison-Wesley, 1988.

[Ghosh 98] Ghosh, Anup K.; O'Connor, Tom; & McGraw, Gary. "An Automated Approach for Identifying Potential Vulnerabilities in Software," 104-114. *Proceedings of the 1998 IEEE Symposium on Security and Privacy*. Oakland, California, May 3-6, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[Gong 99] Gong, Li. *Inside Java 2 Platform Security*. Reading, MA: Addison Wesley, 1999.

[Graff 03] Graff, Mark G. & Van Wyk, Kenneth R. *Secure*

191. http://csrc.nist.gov/nissc/1998/papers.html

Coding: Principles and Practices. Sebastopol, CA: O'Reilly, 2003.

[Hoglund 04] Hoglund, Greg & McGraw, Gary. *Exploiting Software : How to Break Code*. Boston, MA: Addison-Wesley, 2004.

[Howard 00] Howard, Michael. *Designing Secure Web-Based Applications for Microsoft Windows 2000*. Redmond, WA: Microsoft Press, 2000.

[Howard 02] Howard, Michael & LeBlanc, David C. *Writing Secure Code*, 2nd ed. Redmond, WA: Microsoft Press, 2002.

[Jones 91] Jones, Capers. Applied Software Measurement: Assuring Productivity and Quality. New York, NY: McGraw-Hill, 1991.

[Jones 94] Jones, Capers. Assessment and Control of Software Risks. Englewood Cliffs, NJ: Yourdon Press, 1994.

[Jones 86] Jones, Capers. Programming Productivity. New York, NY: McGraw-Hill, 1986.

[Kitson 87] Kitson, David H. & Masters, Stephen. "An Analysis of SEI Software Process Assessment Results, 1987-1991," 68-77. Proceedings of the Fifteenth International Conference on Software Engineering. Baltimore, Maryland. May 17-21, 1993. Washington, DC: IEEE Computer Society Press, 1993.

[Kuperman 99] Kuperman, Benjamin A. & Spafford, Eugene. *Generation of Application Level Audit Data via Library Interposition*. CERIAS Tech Report TR-99-11, 1999.

[Maguire 93] Maguire, Steve. *Writing Solid Code: Microsoft's Techniques for Developing Bug-Free C Programs*. Redmond, WA: Microsoft Press, 1993.

[McConnell 93] McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Redmond, WA: Microsoft Press, 1993.

[McGraw 99] McGraw, Gary, & Felten, Edward W. *Securing Java: Getting Down to Business with Mobile Code*, 2nd ed. New York, NY: John Wiley & Sons, 1999.

[McGraw 02] McGraw, Gary. "Managing Software Security Risks." *Computer 35*, 4 (March 2002): 99-101.

[McGraw 03] McGraw, Gary. "From the Ground Up: The DIMACS Software Security Workshop." *IEEE Security and Privacy 1*, 2 (March-April 2003):

59-66.

[McGraw 04a] McGraw, Gary & Potter, Bruce. “Software Security Testing.” *IEEE Security and Privacy 2*, 5 (September-October 2004): 81-85.

[McGraw 04b] McGraw, Gary. “Software Security.” *IEEE Security and Privacy 2*, 2 (March-April 2004): 80-83.

[McGraw 04c] McGraw, Gary. “Application Security Testing Tools: Worth the Money?[192]” *Network Magazine*, November 1, 2004.

[Miller 90] Miller, Barton P. “An Empirical Study of the Reliability of UNIX Utilities.” *Communications of the ACM 33*, 12 (1990).

[NCSA 97] National Center for Supercomputing Applications. *NCSA Secure Programming Guidelines*, 1997.

[Peikari 04] Peikari, Cyrus & Chuvakin, Anton. *Security Warrior*. Sebastopol, CA: O’Reilly, 2004.

[Saltzer 75] Saltzer, Jerome H. & Schroeder, Michael D. “The Protection of Information in Computer Systems.” *Proceedings of the IEEE 63*, 9 (September 1975): 1278-1308.

[Sessions 03] Sessions, Roger. *Software Fortresses: Modeling Enterprise Architectures*. Boston, MA: Addison-Wesley, 2003.

[Soo Hoo 01] Soo Hoo, Kevin; Sudbury, Andrew W.; & Jaquith, Andrew R. “Tangible ROI through Secure Software Engineering.” *Secure Business Quarterly 1*, 2 (2001).

[Spafford 89] Spafford, Eugene H. “Crisis and Aftermath.” *Communications of the ACM 32*, 6 (1989).

[Spafford 95] Spafford, Eugene H. *UNIX and Security: The Influences of History. Information Systems Security*. Auerbach Publications, 1995.

[Sun 00] Sun Microsystems. *Security Code Guidelines*[193], 2000.

[Swanson 96] Swanson, Marianne & Guttman, Barbara. *Generally Accepted Principles and Practices for Securing Information Technology Systems*. National Institute of Standards and Guidelines Computer Security Special Publication 800-14, 1996.

192. http://www.networkmagazine.com/showArticle.jhtml?articleID=49901410

193. http://java.sun.com/security/seccodeguide.html

[Swiderski 04] Swiderski, Frank & Snyder, Window. *Threat Modeling*. Redmond, WA: Microsoft Press, 2004.

[Thompson 84] Thompson, Ken. "Reflections on Trusting Trust." *Communications of the ACM 27*, 8 (August 1984).

[Viega 00] Viega, John; McGraw, Gary; Mutdoseh, Tom; & Felten, Edward W. "Statically Scanning Java Code: Finding Security Vulnerabilities." *IEEE Software 17*, 5 (September-October 2000): 68-77.

[Viega 01] Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley Professional, 2001.

[Viega 03] Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN 0596003943).

[Voas 97] Voas, Jeffrey & McGraw, Gary. *Software Fault Injection: Inoculating Programs Against Errors*. New York, NY: John Wiley & Sons, 1997.

[Whittaker 04] Whittaker, James A.; Thompson, Herbert H.; & Thompson, Herbert. *How to Break Software Security*. Boston, MA: Addison Wesley, 2004 (ISBN 0321194330).

[Yoder 98] Yoder, Joseph & Barcalow, Jeffrey. "Architectural Patterns for Enabling Application Security[194]." *Proceedings of the 1997 Pattern Languages of Programming Conference*. Monticello, Illinois, Sept. 3-5, 1997. Washington University Technical Report (wucs-97-34), 1998.

Cigital, Inc. Copyright

194. http://st-www.cs.uiuc.edu/~hanmer/PLoP-97/Proceedings/proceedings.zip

1. mailto:copyright@cigital.com

Fields

Name	Value
Copyright Holder	Cigital, Inc.

Fields

Name	Value
is-content-area-overview	true
Content Areas	Tools/Code Analysis
SDLC Relevance	Testing
Workflow State	Publishable